LLNL-TR-609233

I/O Forwarding on Livermore Computing Commodity Linux Clusters

Jim E. Garlick

December 26, 2012

I/O Forwarding on Livermore Computing Commodity Linux Clusters

Jim Garlick, garlick@llnl.gov

December 26, 2012

1 Overview

This report takes the first step in an investigation of I/O forwarding for commodity Linux clusters to improve job launch scalability, reduce operating system jitter, and simplify system management. We discuss the motivation and challenges presented by I/O forwarding through the lens of our experiences with the IBM Blue Gene systems. We identify three I/O modes that each present unique I/O forwarding requirements, and may fare differently in a cost/benefit calculation when considering whether to forward or not. Finally, we present a set of requirements, broken down by I/O mode, which can be used as the basis for an evaluation of possible options.

2 Motivation and Challenges

The original IBM Blue Gene/L system deployed at LLNL in 2004 demonstrated a novel method of scaling I/O to very large compute node counts. IBM recognized that a system presenting 100,000 or 1,000,000 clients to HPC file systems of the day would probably not function, and certainly would not perform well. Thus, instead of mounting file systems directly on each compute node, the Blue Gene architecture, depicted in Figure 1, mounts file systems only on special purpose I/O nodes which proxy I/O requests on behalf of a block of compute nodes, on the order of 64 or 128. The machine thus presents a couple of orders of magnitude fewer file system clients to the site's file system servers.

The Blue Gene architecture has some benefits and drawbacks. Among the benefits are:

- Avoiding $N:1$ network connections to file system servers where N grows very large.

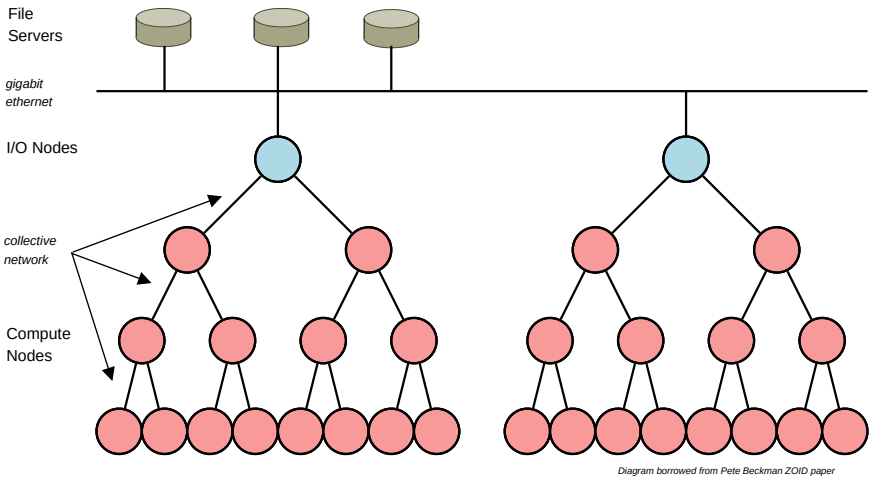

Figure 1: Blue Gene I/O nodes proxy I/O requests on behalf of compute nodes.

- Reducing the number of concurrent requests that must be handled by servers, due to client request aggregation and throttling.
- Reducing the amount of per-client state that must be maintained by servers, thus reducing server memory footprint and server-side processing overhead.
- Reducing the memory and CPU impact of file system clients on compute nodes (there is little to spare on Blue Gene compute nodes).
- In case of parallel read of the same file/directory, shared dentry and page cache on I/O nodes reduces the number of requests sent to servers.

However, the architecture falls short of fulfilling its potential in the Blue Gene environment due to:

- Blue Gene I/O nodes are small embedded systems like the compute nodes and have limited memory, so performance is sometimes limited by I/O node capability, and benefits that might have been obtained through I/O node caching are reduced due to memory pressure on the cache.
- The workload that the file system client on the I/O node must handle looks quite different from the simpler one normally seen when the client is run on a compute node and thus may confound read-ahead, write aggregation, request queuing strategies, or other client side optimizations placed there for a presumed common usage.
- The embedded I/O node hardware and software provided by IBM is challenging to integrate and test with LC file systems and networking.
- The user mode function shipping implementation on Blue Gene creates some special challenges in maintaining POSIX in the compute environment.
- The mapping of I/O nodes to compute nodes on Blue Gene is static, thus a set of jobs space sharing the system and performing I/O in unsynchronized bursts will leave some I/O nodes idle while others are fully utilized.

The Blue Gene architecture has served as a model for other I/O forwarding approaches, such as Cray's DVS[4], Argonne's IOFSL[1], CEA's NFS-Ganesha[2], and Livermore's diod[3]. To date, I/O forwarding has primarily been employed on capability-class machines. However, due to their high productivity, commodity Linux clusters such as the TLCC systems have proliferated and increased in size in recent years, and are starting to present I/O scalability challenges of their own. It is thus natural to consider whether the Blue Gene model can be utilized to any advantage on commodity Linux clusters.

I/O forwarding on Linux clusters will have similar benefits and challenges as on Blue Gene, although the commodity hardware and open software model provides more flexibility to tune the environment. Further, building up a new capability from scratch presents an opportunity to learn from our experiences with Blue Gene and try to improve upon it.

3 I/O Modes

The three categories of I/O listed in Table 1 should be considered when discussing I/O forwarding on commodity Linux clusters. Each mode presents unique requirements, and fares differently in a cost/benefit calculation. It is possible to implement I/O forwarding for some modes but not others, or with appropriate resource manager integration, allow users to choose direct mounts vs forwarding for specific file systems, or configure custom forwarding topologies.

Conspicuously absent from this list is MPI-IO or other I/O middleware usage modes. Much has been written about the opportunities for optimizing parallel I/O when applications expose more of their strategy via API's designed for parallel I/O. Despite its availability for many years, I/O

Mode	File System	LC Usage
POSIX-serial (static, ro)	NFS, ext4 on shared blkdev	root, /usr/local
POSIX-serial (dynamic, rw)	NFS	home, /usr/gapps, /usr/global, NFS tmp
POSIX-parallel (dynamic, rw)	Lustre, Panasas, Ceph, GPFS	simulation data

Table 1: I/O Modes Considered for I/O Forwarding

middleware technology has to a large extent been ignored by LC's user community. Thus, to have an impact in LC, a proposed I/O forwarding mechanism for Linux clusters must focus on POSIX, without precluding the use of such middleware.

3.1 POSIX Serial (static, read-only)

Root and /usr/local file systems at Livermore are updated with packaging discipline and thus can be updated using processes that do not require live read-write access. Because of this, the content in these file systems could be aggressively cached within the system. Currently, copies of these file systems are replicated on each cluster scalable unit (on disk) and direct-mounted read-only on compute nodes via NFSv3. NFSv3 presents a challenge to path search performance and has other undesirable caching properties, such as the need to revalidate cached attributes periodically, despite the fact that the file system content may be static.

Simply introducing a layer of I/O forwarding in front of NFS as on Blue Gene could help. When compute nodes are loading the same data, the I/O node page and dentry caches will be hot, thus the server will see a reduction in requests proportional to the I/O node to compute node ratio.

If we can accept that file system content cannot change while mounted, then we can dramatically improve path search performance and scalability beyond the simple I/O forwarding case by using I/O forwarding in combination with a network block device or loopback mount. In this scheme, file system content is stored in an immutable local file system image (such as EXT4), which is in turn stored in NFS. The NFS file system containing the image file is mounted on I/O nodes as on Blue Gene, but the compute nodes mount the image file directly as though it were a local block device.

The main advantage of this technique is that compute nodes are able to cache metadata via the buffer cache, thus metadata operations are no longer a barrier to scalability. A preliminary study of this approach and its effect on path search is presented in Appendix A.

3.2 POSIX Serial (read-write)

Unmanaged content that is expected to be accessed (mostly) serially, such as home directories, resides on global NFS servers. This content is expected to be in a state of constant change. Global NFS servers are currently direct-mounted on all compute nodes of all commodity Linux clusters. Although designed for serial access, these file systems do present scalability challenges for clusters since they may contain, among other things, the current working directory of running jobs, executables that may be parallel-loaded during launch, core dumps, log files, and input decks.

As with the POSIX serial (static, read-only) mode, a layer of I/O forwarding in this mode allows parallel loaded executables and data to utilize the shared I/O node page and dentry caches and the server will see a reduction of I/O requests proportional to the I/O node to compute node ratio.

3.3 POSIX Parallel

Each network zone contains several Lustre file systems that are direct mounted on all commodity compute nodes within that zone. Lustre contains unmanaged bulk simulation data, treated by the

center as short-term scratch data. We generally think of our workload as being primarily file-per-process parallel I/O, coming in bursts from this job or that job at any given time such that file system bandwidth is amortized across all the jobs space sharing center resources.

The pluses and minuses of Blue Gene I/O forwarding described earlier apply here. A few of the more significant benefits and costs for Lustre are discussed briefly below:

(+) **Reduce Compute Node Jitter and Memory Impact** Each Lustre client maintains state for all Lustre servers, pings LNET peers (all neighboring routers) at T=50s, pings storage targets (usually multiple per OST) at $T = 150s$, and holds locks that are lazily reclaimed. Moving the client off of the compute nodes would have a significant impact on the amount of memory available to applications and would reduce OS jitter. This impact grows proportionally with the number of Lustre servers in the center.

(+) **Reduce or Eliminate LNET Routing** The LNET routing capability introduced flow control issues that sometimes are resolved by LNET requests being dropped. This violates an invariant in the upper layers of Lustre's protocol stack, resulting in myriad unplanned-for failure modes when routers or interconnect fabrics become congested. I/O forwarding could be implemented such that the forwarding nodes are direct-connected (via one LNET hop) to Lustre servers.

(+) **Reduce Distributed Failure** Lustre can be thought of as a big distributed system made up of Lustre clients, servers, and sometimes routers. Lustre QA is weakest in the area of at-scale distributed failure, so an architecture that reduces Lustre client failures ought to avoid tickling a certain class of bugs that will perpetually resurface in the Lustre code base. Clients ought to fail less in the I/O forwarding architecture because:

- There are fewer clients, so randomly distributed failures (e.g. hardware) have less overall effect.
- I/O nodes run more homogeneous activity than compute nodes which are under user control, thus ought to run more predictably; for example, they should be less likely to enter memory reclaim if provisioned properly.
- I/O nodes can be physically located closer to Lustre servers, thus failures and congestion in cluster interconnects are avoided, increasing client reachability.

Caveat: one could expect a decrease in reliability of a single client presented with an I/O forwarding workload, compared with a single client running a single compute node workload.

(−) **Performance** As described above, an I/O node client gets a rather different workload than a compute node client, and we know from Blue Gene experience that Lustre does not necssarily handle it well. It is speculated that to reach performance parity with direct mounts is a 2y effort with current Lustre staff and plans (Chris Morrone).

(−) **Fault Isolation** When each Lustre client handles at most one job, it's simple to correlate Lustre errors with a particular job, and thus track down a user or code that may be causing problems. In addition, a failure in the Lustre client only takes out one job. With I/O forwarding, the situation could be made more awkward if multiple jobs are sending traffic through a single I/O node.

4 Requirements

1. **Common to All Modes**	
1.1.	**POSIX** Forwarding client shall provide identical POSIX semantics to the file system being forwarded. A POSIX file system test suite such as fstest run on I/O node and compute node shall produce identical results.
1.2	**no cache mode** Forwarding client shall support a mount option which disables all client side caching.
1.3	**basic fault tolerance** Active-active fail-over shall be supported across at least two I/O nodes. Network errors resulting from I/O node reboot or loss of an I/O node shall trigger a recovery mode in which the forwarding client shall reconnect to the same or another equivalent server, causing all client side operations to block until a connection is established. In basic mode, all pending requests may be dropped and pending system calls associated with those requests may return an appropriate error; however the mount point remains valid and new requests are handled in the context of the new connection.
1.4	**fault tolerance with replay** Same requirements as basic mode, but upon reconnect, client re-establishes any session state, including open files, and pending operations are retried in the context of the new connection. Recovery design shall favor simplicity, but to the extent possible, minimize any errors returned to users and ensure data integrity. All recovery events shall be logged in such a way that users can be notified (e.g. through resource manager integration).
1.5	**security** I/O forwarding protocol shall provide authentication, integrity, and (optionally) privacy. The security design must include the possibility that valid Kerberos credentials may be required to access NFSv4 shares or Lustre.
1.6	**job separation** It shall be possible for the resource manager to launch I/O forwarding servers dedicated to a particular job, such that they run with only the user's credentials. Where these servers execute shall be determined dynamically by the resource manager.
1.7	**monitoring** A well designed plugin API shall permit the resource manager to collect I/O performance data and event logs pertaining to a particular job.
1.8	**performance** Server shall be capable of handling a number of requests per second comparable to that of its backing file system. Protocol shall minimize the number of requests between client and server and pipeline them wherever possible.
2. **POSIX serial (static, ro)**	
2.1	**network block device** A large file accessible on I/O nodes via I/O forwarding protocol shall be mountable as a block device on compute nodes.
2.2	**no revalidate cache mode** I/O forwarding client shall support a mount option which enables aggressive caching of directory entries and blocks without revalidation or time-out.
2.3	**bootstrap** It shall be possible to mount root via I/O forwarding client from a thin bootstrap image such as a dracut initramfs.
3. **POSIX serial (dynamic, rw)**	
3.1	**NFS CTO cache mode** Forwarding client shall support a mount option which enables caching of directory entries and blocks that is consistent with the NFS close-to-open (CTO) cache consistency model. CTO says that a close system call on one client causes the file attributes and content to be immediately visible when another client issues an open system call.

4. **POSIX parallel**	
4.1	**should not interfere with Lustre consistency** When run in no cache mode (req 1.2) the usual ior, simul, and mdtest tests normally used to test Lustre should produce identical results as regards consistency and correctness with I/O forwarding in place. Large I/O requests up to at least 1MiB shall not be broken up into smaller requests by I/O forwarding system, as doing so might jeopardize record-level consistency.
4.2	**Bulk I/O Performance** Write requests greater than a configurable threshold shall be server-scheduled to avoid overwhelming I/O node buffers. RDMA shall be employed if performance significantly benefits from it.

A Optimizing Path Search in Livermore Computing

Path search is a common UNIX programming idiom and file system use case. It is used in the following cases, to name a few:

- shell or execvp(3)'s search for executables by iterating over PATH elements
- ld-linux.so search for shared libraries by iterating over /etc/ld.so.conf or LD_LIBRARY_PATH elements (although this is mitigated somewhat by ld.so.cache)
- python dynamic linking and loading as explored by the Pynamic Benchmark
- perl module load path

Path Search Example Take for example a shell script that contains a line "hostname". On hype the system default path is set to:

```
/usr/lib64/qt-3.3/bin:/usr/global/tools/totalview/m/hype/dflt/bin
:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin
```

In order to find "hostname", the shell must perform a system call such as stat(2) or execve(2) that will trigger path resolution, as described in path_resolution(7), for each possible path starting with /usr/lib64/qt-3.3/bin/hostname and ending when hostname is found in /bin/hostname.

Cache Effects The dcache assists with this process. For each directory searched unsuccessfully for hostname, a negative dentry is instantiated in the dcache for hostname in that directory. For /bin, a positive dentry is instantiated for hostname in that directory. The next time any process on the system does a path lookup for hostname, it will still need to walk each directory in the PATH but the path resolution for hostname will be short circuited by the existance of the dcache entries, provided the entries continue to meet the underlying file system's criteria for avoiding revalidation. In other words, the underlying file system will not have to contact its network server (NFS, Lustre) or access the block layer (ext3, other local file systems) for every hostname path search.

However, It is worth noting that the dcache caches names not directory blocks. Therefore the dcache cannot ever satisfy a request for a file name that has not been requested before. So even though only a handful of files exist in /usr/lib64/qt-3.3/bin every path search for a new name begins by triggering a lookup in the file system backing /usr/lib64/qt-3.3/bin for the new name. For local file systems this is not too bad because the block layer caches disk blocks backing the directory, and the underlying file system will still likely be able to satisfy the lookup via the buffer cache (in local memory). For network file systems, this lookup triggers an RPC to the network file system server.

PATH Search for Executables on LC Systems On our systems all of the above PATH components are on network file systems! When a new name is resolved, the latency is the sum of the RTT's of an RPC to each server (requests are not pipelined). The root and /usr/local file systems are read-only, with replicas served from RPS nodes in each scalable unit. The /usr/global file system is a single enterprise NFS server shared center-wide.

Configuration	Wall clock	Notes
Direct ext4 on SATA disk on Opteron	3.375s	
The same file system mounted on Atom with NFSv3	40.515s	rpcinfo counted 160,485 RPC's to NFS server during test
Loopback mount ext4 image on Opteron	2.896s	
Loopback mount ext4 image on Atom, image in 9p	5.475s	
Loopback mount ext4 image on Atom, image in NFSv3	5.163s	
NBD mount on Atom	4.842s	
iSCSI mount on Atom	6.717s	

When path search is invoked during parallel job launch, the load on shared servers multiplies and RTT degrades. For distant, non-replicated servers, the RTT can become very large.

Improving LC PATH Search Performance The system default PATH should contain only essential entries, and global file systems especially should be removed if possible. The following are likely non-essential for most users: /usr/lib64/qt-3.3/bin, /usr/local/sbin, /usr/sbin, and /sbin.

Totalview should be packaged for /usr/local like all the other DEG software development environment tools and /usr/global/tools/totalview/m/hype/dflt/bin removed from the path. This will remove the most egregious RTT scalability problem from the PATH.

For file systems like root and /usr/local that can be used read-only, consider replacing NFS with a local file system like ext4, shared (read-only) using a network block device implementation, as described in root file system on network block device. This brings the buffer cache to bear on directory lookups for previously unknown file names and could improve scaling (and indeed it does, see below).

Todo: Investigate NFSv4 directory delegations

What about I/O forwarding? In theory I/O forwarding should help this case because the dcache on forwarding nodes is shared. In practice, diod/9p forwarding disables the dcache on compute nodes and the net benefit for path lookup is negative for forwarded NFS. We have not tried with 9p's caching enabled (because it changes expectations of NFS cache coherence).

Testing NFS versus Network Block Device To verify the claim that path search performance is better on a local file system atop a network block device compared to NFS, a small test was arranged.

Test setup: 1g ext4 image containing 16 directories with 10,000 zero length files each. The pathwalk test, for each of 10,000 file names, iterates through the 16 directories trying to stat the name in each directory. For each name, the first 15 attempts will fail and the 16th will succeed. Caches were dropped between tests. The following table shows wall clock times for the test between one Opteron server node and one Atom client:

These results illustrate how NFS's caching strategy yields poor path search performance compared to ext4 backed by a network block device.

There is insufficient data to conclude anything about which of the four network block device modes is better. Other network block device methods we could test: SRP, iSER, GNBD, and RADOS.)

The same test was run on 82 nodes of hype on NFSv3 (server: local RPS node), and on an ext4 image stored in /g/g0 (global NFS), forwarded to compute nodes with 9p/diod, and loopback mounted on the compute nodes. Caches were dropped between tests. The resulting graph of wall-clock run times of pathwalk on various node counts, shows that while NFS offers poor scaling, ext4 backed by a network block device scales essentially the same as the "null job" on hype. Todo: mpi version of pathwalk so launch time is already factored out).

Note that the NFS curve is expected to continue up with the node count for a global NFS server like /usr/global or /usr/gapps. For replicated, read-only file systems like /usr/local or root, the

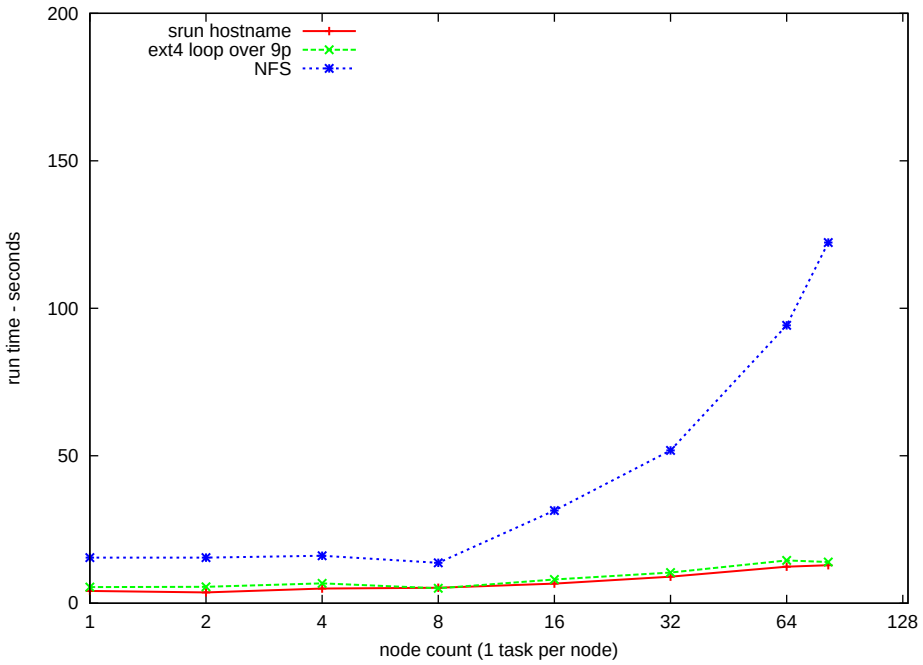

Figure 2: Path walk performance for distributed block device based on 9P and diod scales the same as the null slurm job up to 82 nodes, while NFS begins to slow at 8 nodes.

NFS curve tops out at the scalable unit ratio of compute to RPS server nodes (about where this graph does, for our systems).

Multiple tasks per node were not attempted. The scaling results should approximate the single node results above in both NFS and network block device cases due to dcache effects.

Future Directions Applications that live in global NFS file systems really need to be moved to a medium more suited to parallel launch.

LC could provide a tool set to users and code teams for composing an immutable file system image that contains all the executable and configuration data needed to run. These images could be stored forever in an LC copy-on-write snapshot service of some type under unique identifiers. One would provide the identifier as part of the job submission, and the resource manager would get the image mounted (read-only) on all compute nodes working on the job. As a side benefit, this executable image could become part of the job's execution record and be pulled out later on for another run in case results need to be rerun or checked. In fact, /usr/local and perhaps root images could utilize this same service, with versions selectable by users on a per-job basis.

References

[1] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. A scalable I/O forwarding framework for high-performance computing systems. Technical Report OSU-CISRC-4/09-TR07, Ohio State University, 2009.

[2] P. Deniel. Nfs-ganesha. Retrieved Dec. 26, 2012 from `http://sourceforge.net/apps/trac/nfs-ganesha/`.

[3] J. Garlick. diod distributed I/O daemon. Retrieved Nov. 9, 2012 from `http://code.google.com/p/diod/`.

[4] S. Sugiyama and D. Wallace. Cray DVS: Data virtualization service. *Cray User Group (CUG 2008)*, 2008.